Army Research Laboratory

Automating RPM Creation from a Source Code Repository

by Travis Parker and Paul Ritchey

ARL-CR-0687 **February 2012**

Prepared by
ICF International
9300 Lee Highway,
Fairfax, VA 22031-1207 USA

Under Contract W911QX-07-F-0023

COR:
U.S. Army Research Laboratory
RDRL-CI
Linda Duchow
2800 Powder Mill Road
Adelphi, MD 20783-1197

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-CR-0687 **February 2012**

Automating RPM Creation from a Source Code Repository

Travis Parker and Paul Ritchey
ICF International

Prepared by
ICF International
9300 Lee Highway
Fairfax, VA 22031-1207 USA

Under Contract W911QX-07-F-0023

COR:
U.S. Army Research Laboratory
RDRL-CI
Linda Duchow
2800 Powder Mill Road
Adelphi, MD 20783-1197

REPORT DOCUMENTATION PAGE						*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**						
1. REPORT DATE *(DD-MM-YYYY)* February 2012		**2. REPORT TYPE** Final				**3. DATES COVERED (From - To)**
4. TITLE AND SUBTITLE Automating RPM Creation from a Source Code Repository						**5a. CONTRACT NUMBER** W911QX-07-F-0023
						5b. GRANT NUMBER
						5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Travis Parker and Paul Ritchey						**5d. PROJECT NUMBER**
						5e. TASK NUMBER
						5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ICF International 9300 Lee Highway Fairfax, VA 22031-1207 USA						**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-CIN-D Aberdeen Proving Ground, MD 21005						**10. SPONSOR/MONITOR'S ACRONYM(S)**
						11. SPONSOR/MONITOR'S REPORT NUMBER(S) ARL-CR-0687
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT The use of a source code repository while writing and maintaining a software project has become a modern standard. With careful planning and standardization of the structure of the source code repository, it is possible to automate the construction of Red Hat Package Manager packages (RPMs). This automation encompasses checking out the most recent release of the software, compiling the software, if necessary, to create binaries through building the actual RPM that can then be immediately used to install the software on a computer. This report describes the repository structure and script we developed to allow us to automate this process.						
15. SUBJECT TERMS RPM, software development, source code repository						
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**		**19a. NAME OF RESPONSIBLE PERSON** Travis Parker
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	26		**19b. TELEPHONE NUMBER** (*Include area code*) (410) 278-0900

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures **iv**

1. Introduction **1**

2. Source Code Repository Structure **1**

3. Example Repository Structure **2**

4. Tagging Release Versions and Version Naming **3**

5. Automated RPM Build Basic Project Setup **4**

6. RPM Specification File Overview **4**

7. RPM Specification File Tag Examples **7**

8. Performing an Automated RPM Build **9**

9. Conclusion **10**

Appendix A. Release Tagging Script `tag-it.sh` **11**

Appendix B. The `build_rpm.sh` Script **13**

Appendix C. Sample RPM Specification File **17**

Distribution List **19**

List of Figures

Figure 1. Fictitious example of a repository structure. 2
Figure 2. Template specification file 5
Figure 3. Sample code from a C/C++ project that supports the GNU AutoTools compilation process....................7
Figure 4. Sample code with the global permission setting matrix used....................8
Figure 5. Sample `<change_log>` code. 8
Figure 6. Sample content used to define several environment variables needed by the `build_rpm.sh` script. 9

1. Introduction

The use of a source code repository while writing and maintaining a software project has become a modern standard. With careful planning and standardization of the structure of the source code repository, it is possible to automate the construction of Red Hat Package Manager[1] packages (RPMs). This automation encompasses checking out the most recent release of the software, compiling the software, if necessary, to create binaries through building the actual RPM that can then be immediately used to install the software on a computer. This report describes the repository structure and script we developed to allow us to automate this process.

2. Source Code Repository Structure

For the purposes of this report, the provided scripts are based on using Subversion[2] for the source code repository. The structure and concepts provided should be easily implemented if other source code repositories are used, such as CVS[3].

Each major project or component that should be built into a package will need to contain a specific directory structure that will be used to maintain the current working branch, tagged release versions, and alternate branches:

- `trunk` – This directory contains the current code under development and is typically used by all developers working on the project.
- `branch` – This directory contains an alternative set of code that may be under development. This is typically used to "fork" the primary development code to work on a new version that is not yet ready to replace the primary code.
- `tag` – This directory contains specific revisions from the primary development code ("trunk") that have been marked as release versions. This is the directory that the automated RPM build procedure will check out code from when building RPMs. No development work or commits are performed against tagged revisions, they are considered final releases.

[1]Foster-Johnson, E. *RPM Guide*. Fedora Documentation. Fedora Project. Web. 31 May 2011. http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/index.html (accessed November 2011).

[2] Collins-Sussman, B. *Version Control with Subversion*. O'Reilly Media, 2008. Web. 31 May 2011. http://svnbook.red-bean.com/ (accessed November 2011).

[3] Price, D. R. *CVS - Concurrent Versions System*. CVS - Open Source Version Control. Free Software Foundation, 3 Dec. 2006. Web. 31 May 2011. http://www.nongnu.org/cvs/ (accessed November 2011).

The structure within a project or component is up to the developer(s) working on it, but if the code is written in C/C++ the structure outlined in the report titled *Simple Guide to Using GNU AutoTools*[4] is highly recommended. By using the GNU AutoTools for automating the configuration/compilation of one's C/C++ code, the automated RPM build procedure can automatically compile the project when it builds the RPM.

3. Example Repository Structure

The following fictitious example of a repository structure (figure 1) shows how a project can be broken up into several sub-components, each of which contains the proper directory structure so they can be built into separate RPMs.

```
Project_foo
`-- component_a
    |-- subcomponent_a
    |   |-- branch
    |   |-- tag
    |   |   |-- 20100630-1.0-1
    |   |   |-- 20101103-2.0-1
    |   |   `-- 20101203-2.0-2
    |   `-- trunk
    |       |-- SPECS
    |       |-- config
    |       |-- include
    |       |-- src
    |       `-- web
    `-- subcomponent_b
        |-- branch
        |-- tag
        |   |-- 20110119-1.0-1
        |   `-- 20110208-2.0-2
        `-- trunk
            |-- SPECS
            |-- config
            |-- include
            `-- src
```

Figure 1. Fictitious example of a repository structure.

Examining the basic directory structure depicted in figure 1, there have been no new code branches for either of the subcomponents. There have been three production releases for the `subcomponent_a` project, which can be seen by the three directories listed in the `tag` directory. The second project, `subcomponent_b`, has two production releases. Both subcomponents are C/C++ based so the structure within the `trunk` directory includes additional

[4] Parker, T.; Ritchey, R. P. *Simple Guide to Using GNU AutoTools*; ARL-CR-0681; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, October 2011.

subdirectories to support the use of GNU AutoTools. Both subcomponents also support the automated build process, which can be determined by the inclusion of the `SPEC` directory, which is described later.

4. Tagging Release Versions and Version Naming

Before describing the automated RPM mechanism that has been built, a brief discussion of the convention used for version naming is needed. A specific format must be used for the naming convention when tagging release versions in the `tag` directory. This naming scheme is relied upon by the automated RPM build script when constructing the name of the RPM and when it automatically fills in text in the specification file.

The name is constructed of four sections, separated by either a dash (-) or a period (.). A side benefit to following this convention is the versions will automatically be listed in a sorted order, making it easy to determine which one is the most recent release when performing a checkout by hand.

- `<YYYYMMDD>-<major_version_number>.<minor_version_number>-<revision_number>`

  Example: `20100630-2.0-1`

  This is the date the release/tag is made using four digits for the year (YYYY), two digits for the month (MM), and two digits for the day (DD). The month and day should be preceded with a zero if necessary to make them two digits each.

- `<major_version_number>.<minor_version_number>`

  This is a typical software version numbering scheme using a major/minor indicator. Packages whose software has received minor updates (new features, enhancements, or fixes) should increment the minor version number value and packages whose software has received some type of major change should have the major version number incremented.

- `<revision_number>`

  The revision number indicates either an update to the package or some type of very minor update to the software. The revision number can be one of two numbers, which is up to the discretion of the developer/maintainer. It can be the revision number of a specific commit (save) in the code repository or it can be a manually maintained and incremented number.

A script that can be used to automate the process of tagging releases is provided in appendix A. The script defaults to using the repository commit number if a revision number isn't provided when the script is executed.

5. Automated RPM Build Basic Project Setup

To ease the administration and maintenance of large projects, a convention has been defined for incorporating the files necessary to build an RPM non-intrusively into the source code. When this convention is followed, it allows one to build an RPM package for a project, component, or subcomponent by using the generic `build_rpm.sh` script (provided in appendix B). For projects where multiple packages are desired, multiple calls to the `build_rpm.sh` script can be combined into a single master script that when used will automatically build *all* packages needed for that project.

In the component/subcomponent's `trunk` directory create a `SPECS` subdirectory (all capital letters), which will contain one file, the specification file for building the RPM. The name of the specification file must adhere to the following format for the script to function:

```
<package_name>.spec
```

The <package_name> portion should be set to the name one wants used for the RPM package file without the version or revision numbers. When the RPM package is built, the script will automatically determine and append the version number and revision number based on the tagged name contained in the `tag` directory.

For example, if the automated build script is called to build the `subcomponent_a` package based on the listing provided in the previous section, a valid package name could be the following:

```
projecta_componenta_subcomponenta-2.0-2.x84_64.rpm
```

The beginning portion of the name is taken from the name of the specification file (`projecta_componenta_subcomponenta.spec`), the version number is derived from the repository tagged name and the platform ('`x84_64`') is taken from the contents of the specification file (`BuildArch` entry), which is described in section 6.

6. RPM Specification File Overview

To build an RPM, one needs to provide a specification file[5] (generally referred to as a "spec file") that provides the `rpm` command with the information it needs to build the package. All of the specification files can be derived from the same basic template, changing a few specific lines

[5] "Creating the Spec File." Maximum RPM: Taking the Red Hat Package Manager to the Limit. Web. 31 May 2011. <http://www.rpm.org/max-rpm/s1-rpm-build-creating-spec-file.html>.

that are unique for each project. Figure 2 shows a template specification file; a completed sample specification file can be found in appendix C.

```
Summary: <summary_description>
Name: %NAME_MARKER%
Version: %VERSION_MARKER%
Release: %RELEASE_MARKER%
License: <license>
Group: <group_name>
Source: %SOURCE_MARKER%.tgz
Distribution: <distribution_information>
Vendor: <vendor>
Packager: <packager>
BuildRoot: %{_tmppath}/%{name}-buildroot
Provides: %NAME_MARKER%
BuildArch: <target_architecture>
AutoReqProv: <ignore_dependencies>

%description
<complete_description>

%pre

%prep
%setup

%build
<commands_to_compile>

%install
<commands_to_install>

%clean
rm -rf $RPM_BUILD_ROOT

%files
<files_included/installed_by_rpm>

%post

%changelog

<change_log>
```

Figure 2. Template specification file.

While the template in figure 2 can be used as the basis for a specification file for each new project, it is much easier to take a specification file from an existing project and change it instead. Select a specification file from a project that's similar to the new one and use that as the basis for the new project (i.e., pick a specification file from a C/C++ project if the new project is a C/C++ project).

The following is a list of the tagged areas in the sample template above describing what the contents should be changed to and a sample value.

- `<summary_description>` This entry is a free text area where a brief summary (one sentence) describing the package can be placed. A more complete, in-depth description is provided in a different tag.
- `<license>` This shows the type of license the package is released under. A possible example would be `GPL`; however, this will depend on the environment one is developing the package in.
- `<group_name>` This names the group providing the package. An example would be `ARL-SBNAB`.
- `<distribution_information>` Typically, this is set to a text string describing a Linux operating system (OS) distribution. For packages not related to a Linux distribution, this can be set to a value that can be used to group the user's packages or related projects.
- `<vendor>` This text string names who is providing the package. We have been specifying `ARL` here as the work and packages are created under contract for the U.S. Army Research Laboratory (ARL).
- `<packager>` This text string names the person or group packaging the software, for example, `ARL-SBNAB`.
- `<complete_description>` This is a free text area for providing a more complete description of the package than what is provided in the `<summary_description>` tag.
- `<commands_to_compile>` This holds the shell commands required to build a binary out of the code checked out from the repository. This may be left blank or eliminated for projects that do not require compilation, such as scripts (Python, shell, Perl, PHP, etc.). An example for building a C/C++ based project is included in section 7.
- `<commands_to_install>` This holds the shell commands required to install the binaries/scripts. This section is needed at all times, even if the commands to perform a compilation are not needed. An example is included in section 7.
- `<files_included/installed_by_rpm>` This is the list of files, including their path, that are installed by the package. It is also possible to specify default ownership and permissions, which can be overridden on a file by file basis. An example is included in section 7.
- `<change_log>` This holds information describing the changes made in each version/revision. Although partially made up of free text, a specific format must be used so the `rpm` command can properly parse it. An example and description of the format is included in section 7.

Throughout the specification file are additional tags that are marked with starting and ending percent signs (%). These are special markers in the file and should not be changed or altered in any way. The `build_rpm.sh` script will automatically replace these tags with the appropriate text, such as the package name and version number.

7. RPM Specification File Tag Examples

The `<commands_to_compile>` section contains the shell commands that need to be used to configure and compile the source code. The sample in figure 3 is taken from a C/C++ project that supports the GNU AutoTools compilation process, but the concept can be adapted for other compilers and languages. It first executes the `autogen.sh` script to ensure that the macros used by AutoTools have been updated for compatibility with the system the package is being compiled on. It then executes the `configure` script, passing in several optional configuration parameters to enable certain desired features in the resulting binaries. Finally, the `make` command is executed, which performs the compilation. As seen in the provided example, it's possible to combine multiple commands onto a single line just as can be done in a normal shell prompt or script by placing a semicolon (;) between them.

```
./autogen.sh ; ./configure --with-db=/apps/usr --with-
   libpq=/apps/postgres

make
```

```
rm -rf $RPM_BUILD_ROOT
umask 0077
mkdir -p $RPM_BUILD_ROOT/usr/local/bin
mkdir -p $RPM_BUILD_ROOT/usr/local/etc
install -p src/mybinary $RPM_BUILD_ROOT/usr/local/bin
cp -v config/mybinary.cfg $RPM_BUILD_ROOT/usr/local/etc/mybinary.cfg
```

Figure 3. Sample code from a C/C++ project that supports the GNU AutoTools compilation process.

The `<commands_to_install>` section contains the shell commands that are needed to install the binaries/scripts for the package. Every specification file should include the same first line, which cleans up the directory where the RPM is being constructed. The rest of the commands will be highly customized for each project. If another project's specification file is used, this section will need to be carefully reviewed to make sure each line is performing a task that is valid and needed to install that particular package. The location the files should be installed to are based off of the `$RPM_BUILD_ROOT` environment variable, with the rest of the path being the exact path the file will be installed in on a real system. In the C/C++ example in figure 3, a compiled binary is copied from the source code `src` directory and a default configuration file is copied as well.

The `<files_installed/included_by_rpm>` section should contain an exhaustive list of all files, including their path, that are installed by the RPM. Additional macros such as `%defattr()` can be used to set default permissions and ownership for the files as well as being done on a file by file basis. In the example in figure 4, the global permission setting matrix is used, which will be applied to all files/directories installed or created by the RPM.

Note: At the time of writing a known bug with the `%defattr()` macro was causing incorrect ownership and permission settings to be set for directories if they were to be owned by anyone other than the `root` user. A solution to this problem is to use the `%attr()` macro on individual directories.

```
%defattr(750, my_user, my_group, 750)
/usr/local/bin/foo
/user/local/etc/foo.cfg
```

Figure 4. Sample code with the global permission setting matrix used.

The `<change_log>` section of the specification file should be updated when changes are made to the software or package so others have an indication of what has been changed in the new release. The `rpm` command enforces a specific format, which must be used or the package will not be built. The change log entries are broken into two sections, the date/maintainer line which is preceded by an asterisk (*) and the list of changes consisting of one change per line preceded by a hyphen (-), as shown in figure 5.

```
* <day_of_wk> <month> <day_of_mnth> <year> <version_number>
<maintainer> - <one_change>
- <first change>
- <second change>
…
```

Figure 5. Sample `<change_log>` code.

The following apply to code in figure 5:

- `<day_of_wk>` is the three letter abbreviation for the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, or Sat).
- `<month>` is the three-letter abbreviation for the month.
- `<day_of_mnth>` is the two-digit number of the day of the month.
- `<year>` is the year specified in four digits.
- `<version>` can be any string, typically in the format "Version <major>.<minor>".
- `<maintainer>` is the basic information about the person that is creating the new release, such as the person's name or e-mail address.

8. Performing an Automated RPM Build

Once the RPM specification file has been created in the `SPECS` directory and the release has been tagged in the `tag` subdirectory the automated RPM build mechanism can be used. When the `build_rpm.sh` script is executed, it will automatically create an `.rpmmacros` file (note the period at the beginning of the filename) in the home directory. The content of this file is used to define several environment variables needed by the `build_rpm.sh` script and will contain entries similar to that in figure 6.

```
%_topdir /home/myhomedir/rpm
%_tmppath /home/myhomedir/rpm/tmp
%_buildroot ${_tmppath}/%{name}-buildroot
%define _unpackaged_files_terminate_build 0
```

Figure 6. Sample content used to define several environment variables needed by the `build_rpm.sh` script.

The values in the first two lines will automatically be configured to point to the home directory when the file is created. In the example in figure 6, the RPMs will be generated and stored in the `rpm` subdirectory of a user named `myhomedir`.

To execute the `build_rpm.sh` script, one simply passes a single parameter to it indicating the repository of the tagged project release that needs to be checked out and packaged:

```
./build_rpm.sh -r <repository_path>/tag
```

Example:

```
./build_rpm.sh -r file:///apps/rcs/svn/project_a/component_a/subcomponent_a/tag
```

This example will automatically check-out the most recent tagged release for the `subcomponent_a` project, compile it if necessary and create the RPM, which will appear in a subdirectory in `/home/myhomedir/rpm`.

For systems that consist of multiple RPMs maintained as separate projects in the repository, a master script can be created that contains a line for each RPM that needs to be created. This automation makes it easy to build a large number of RPMs by executing one script and walking away while all of the individual projects are checked out, compiled, and turned into installable RPMs.

9. Conclusion

The tools provided here allow developers to easily build installable RPM packages directly from revisions tagged for release in a source code repository. In addition to saving developers a significant amount of time by automating the process, it can provide non-developers a simple, single command for obtaining the latest version of software from the repository without the need to learn how to check software out or compile it.

Appendix A. Release Tagging Script `tag-it.sh`

The `tag-it.sh` script (shown below) can be used to tag releases of a project in a code repository. The script will require minor modifications to adjust the base directory where the repository can be located in the filesystem on the user's server. The script is called in the following format and takes several parameters:

```
tag-it.sh <repo> <major.minor> \"<message>\" [release, next-revision is default]
```

where

- <repo> is the path within the repository to the base directory of the project.
- <major.minor> is the major and minor version number for the release separated by a period.
- <message> is the log message to be used when the release is tagged. The backslash ("\") character is used to prevent the shell from interpreting the quotes.
- [release]: Optional. If the release number is not provided the script will automatically default to using the last commit number for the project.

```
#!/bin/bash
if [ "$1" != "" ]
then
        echo file:///apps/rcs/svn/$1/tag/
        svn list file:///apps/rcs/svn/$1/tag/
fi

if [ "$2" == "" -o "$3" == ""  ]
then
        echo "usage: tag-it.sh <repo> <major.minor> \"<message>\" [release, next-
revision is default]"
        exit 1
fi


if [ "$4" == "" ]
then
        ctr=`svn info file:///apps/rcs/svn/$1 | grep Revision | cut -d' ' -f 2`
        ctr=`expr $ctr + 1`
else
        ctr=$4
fi
echo copy file:///apps/rcs/svn/$1/trunk file:///apps/rcs/svn/$1/tag/`date +%Y%m%d`-$2-
$ctr -m "$3"
svn copy file:///apps/rcs/svn/$1/trunk file:///apps/rcs/svn/$1/tag/`date +%Y%m%d`-$2-
$ctr -m "$3"
echo list file:///apps/rcs/svn/$1/tag/
svn list file:///apps/rcs/svn/$1/tag/

exit 0
```

INTENTIONALLY LEFT BLANK.

Appendix B. The **`build_rpm.sh`** Script

The `build_rpm.sh` script is a generic script that can be used to automatically check out the latest release from a repository and build the RPM. For projects consisting of multiple packages, a master build script can be written that contains multiple calls to the `build_rpm.sh` script, one for each package that needs to be built.

```
#!/bin/bash

# Initialize some variables before we get started.
svnrepo=""

# Make sure script isn't being executed as the root user.
if [ `id -u` -eq 0 ]
then
        echo "ERROR:  RPM build script can NOT be run by root"
        exit 1
fi

# Print a few blank lines so script output is more visible.
echo -e "\n\n\n"

# Process passed in options.  If no parameters are passed, then we'll
# prompt for them.
while getopts "hr:" flag
do
  case $flag in
    h) echo "This script is used to automate building packages maintained in the svn 
repository."
       echo "The following parameters are accepted:"
       echo ""
       echo "    <no parameters>:  Script will prompt you for needed parameters"
       echo ""
       echo "    -h                Print this help text and exit."
       echo "    -r <repo_path>    Repository path to pull source code from."
       echo ""
       echo ""
       exit 0
       ;;
    r) svnrepo=$OPTARG
       ;;
  esac
done

# Create the necessary directories in the user's home directory.
echo "Creating RPM build directories in your home directory...."
for i in BUILD RPMS SOURCES SPECS SRPMS tmp; do
        mkdir -p ~/rpm/$i
done

for i in i686 noarch x86_64; do
        mkdir -p ~/rpm/RPMS/$i
done

# Create the rpmmacro file
echo "Creating rpm macro file (~/.rpmmacros)"
echo "%_topdir $HOME/rpm" > ~/.rpmmacros
echo "%_tmppath $HOME/rpm/tmp" >> ~/.rpmmacros
echo "%_buildroot \${_tmppath}/%{name}-buildroot" >> ~/.rpmmacros
echo "%define _unpackaged_files_terminate_build 0" >> ~/.rpmmacros

# Move to the directory where we will pull checkout the repository to.
cd $HOME/rpm/SOURCES

# If not passed in as a parameter, prompt user for repository containing the software
to be checked out.
if [ "${svnrepo}m" == 'm' ]
then
    read -e -p "Enter the svn repository path to checked out:   " svnrepo
fi

#get the version and release from the tag
if [ `basename ${svnrepo}` == "tag" ]
then
    # The release names in the tag section of the repository are expected to be
    # in the following format:
    # YYYYMMDD-<ver>-<rel>
    #
    # YYYY = 4 digit year
    # MM   = 2 digit month
    # DD   = 2 digit day
    # <ver> = version number (such as '1.0' or '1.0p')
    # <rel> = release number (such as '1' or '3')
    echo "-- Packaging production release."
    release=`svn ls ${svnrepo} | tail -1`
    vernum=`echo ${release} | cut -d'-' -f2`
    relnum=`echo ${release} | cut -d'-' -f3 | sed -e 's/\///g'`

    # Now build the complete svnrepo path for the checkout.
    svnrepo="${svnrepo}/${release}"
fi

# From the repository we can get the package name from the SPEC file
basename=`svn ls ${svnrepo}/SPECS | cut -d. -f1`

# Prompt the user for the rpm package version number.
if [ "${vernum}m" == 'm' ]
then
        read -e -p "Enter the rpm package version number:   " vernum
fi

# Prompt the user for the rpm package release number.
if [ "${relnum}m" == 'm' ]
then
        read -e -p "Enter the rpm package release number:  " relnum
fi

echo "Checking out ${release} from svn...."
svn co ${svnrepo} $basename-$vernum

# Copy the SPEC file from the repository check-out to the proper directory.
cp -v $basename-$vernum/SPECS/$basename.spec ~/rpm/SPECS/$basename-$vernum-
$relnum.spec

# Remove the repository SPECS directory, we don't want it inluded in the RPM.
rm -rf $basename-$vernum/SPECS

# Now remove the hidden .svn directories, we don't want them included in the RPM.
find  $basename-$vernum -type d -name '*.svn' -exec rm -rf '{}' \;
```

```
SUB='$/=undef; $O=`$ARGV[2]`; open(F,$ARGV[0]); $_=<F>; s/$ARGV[1]/$O/g; print $_;'

# Replace the remaining markers in the SPECS file with the actual package name.
cat ~/rpm/SPECS/$basename-$vernum-$relnum.spec.3 | sed -e
"s/%NAME_MARKER%/$basename/g" | sed -e "s/%VERSION_MARKER%/$vernum/g" | sed -e
"s/%RELEASE_MARKER%/$relnum/g" | sed -e "s/%SOU
RCE_MARKER%/$basename-$vernum/g" > ~/rpm/SPECS/$basename-$vernum-$relnum.spec

# Now turn the source directory into a gzipped tarball.
tar zcvf $basename-$vernum.tgz $basename-$vernum

# Now remove the source directory and temporary SPEC files.
rm -rf $basename-$vernum

# Finally build the rpm package.
rpmbuild -bb ~/rpm/SPECS/$basename-$vernum-$relnum.spec
```

INTENTIONALLY LEFT BLANK.

Appendix C. Sample RPM Specification File

The following is a complete specification file.

```
Summary: sample_application: RPM creation test package
Name: %NAME_MARKER%
Version: %VERSION_MARKER%
Release: %RELEASE_MARKER%
License: None
Group: ARL-SBNAB-PITT
Source: %SOURCE_MARKER%.tgz
Distribution: ARL-SBNAB-PITT Tools
Vendor: ARL
Packager: ARL-SBNAB-PITT
BuildRoot: %{_tmppath}/%{name}-buildroot
Provides: %NAME_MARKER%
BuildArch: x86_64
AutoReqProv: no

%description
This package contains a simple hello world application for testing RPM build
from a source code repository.

%pre

%prep
%setup

%build
./autogen.sh ; ./configure --with-db=/apps/db --with-libpq=/apps/postgres
make

%install
rm -rf $RPM_BUILD_ROOT

umask 0077
mkdir   -p $RPM_BUILD_ROOT/usr/local/bin
mkdir   -p $RPM_BUILD_ROOT/usr/local/etc
install -p src/hello $RPM_BUILD_ROOT/usr/local/bin
cp -v config/hello.cfg $RPM_BUILD_ROOT/usr/local/etc/hello.cfg

%clean
rm -rf $RPM_BUILD_ROOT

%files
%defattr(750, root, root, 755)
/usr/local/bin/hello
/usr/local/etc/hello.cfg

%post

%changelog
Place the changelog info for the RPM here
```

NO. OF COPIES	ORGANIZATION

ADMNSTR
DEFNS TECHL INFO CTR
ATTN DTIC OCP
8725 JOHN J KINGMAN RD STE 0944
FT BELVOIR VA 22060-6218

US ARMY RSRCH LAB
ATTN RDRL CIN D T PARKER
BLDG 310E RM C53
ABERDEEN PROVING GROUND MD 21005

US ARMY RSRCH LAB
ATTN RDRL CIN S P RITCHEY
BLDG 310E RM C72
ABERDEEN PROVING GROUND MD 21005

US ARMY RSRCH LAB
ATTN IMNE ALC HRR MAIL & RECORDS MGMT
ATTN RDRL CIO LL TECHL LIB
ATTN RDRL CIO LT TECHL PUB
ADELPHI MD 20783-1197

INTENTIONALLY LEFT BLANK.